CALCULATING FUNCTIONALS OF SOLUTIONS OF LARGE, SPARSE SYSTEMS

by

Trisha R. Butler

B.S., University of Pittsburgh, 2002

Submitted to the Graduate Faculty of

the Department of Mathematics in partial fulfillment

of the requirements for the degree of

Master of Sciences

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

MATHEMATICS DEPARTMENT

This thesis was presented

by

Trisha R. Butler

It was defended on

July 13th 2006

and approved by

Dr. William Layton, Department of Mathematics

Dr. Mike Sussman, Dept. of Mathematics

Dr. David Swigon, Dept. of Mathematics

Thesis Advisor: Dr. William Layton, Department of Mathematics

CALCULATING FUNCTIONALS OF SOLUTIONS OF LARGE, SPARSE SYSTEMS

Trisha R. Butler, M.S.

University of Pittsburgh, 2006

Problems in many applications lead to large, sparse linear systems with coefficient matrices that are invertible and have little other structure. In such problems, the solution $u = A^{-1}f$ is typically calculated only to compute further functionals of that solution. This paper performs preliminary research into the practical question: determine methods that converge to the functional value $l_n \to l(u)$ much more rapidly than $u_n \to u$.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

Problems in many applications lead to large, sparse linear systems with coefficient matrices that are invertible and have little other structure. For example, convective transport problems yield linear systems

$$Au = f,\ A : N \times N,\ \text{invertible matrix} \tag{1.1}$$

where the coefficient matrix A can be large, such as the order of millions for many problems, and at best all that can be said is that the symmetric part of A, $\frac{A+A^*}{2}$, is positive definite. Problems in three body scattering theory lead to linear systems where N can be of the order of 20 million and the coefficient matrix is very sparse but has little other discernible structure. In such problems the solution $u = A^{-1}f$ is typically calculated only to compute further functionals of that solution, upon which decisions are taken. This is obviously necessary–sifting through 20 million data values of equal importance is humanly impossible. The functionals can be complicated linear and nonlinear functionals

$$\text{solve } u = A^{-1}f, \text{ compute } l(u). \tag{1.2}$$

In all cases, however, a very interesting and practical question arises: determine methods that converge very rapidly to the functional value:

$$l_n \to l(u) \text{ much more rapidly than } u_n \to u. \tag{1.3}$$

This is especially important when the coefficient matrix A is not symmetric positive definite and when the lack of discernbile structure in A makes constructing effective preconditioners difficult. This report begins consideration for problem (1.3) in the simplest case in which $l(u)$ is a linear functional. In this case, some simple acceleration formulas for l_n are possible.

Note that if the functional is linear, the problem can be posed as solving the augmented linear system for u and l: Find u, l satisfying

$$l - \langle l, u \rangle = 0, \ Au = f \tag{1.4}$$

If this augmented system is solved by standard iterative methods, no new algortihms arise; this approach is equivalent to computing $l_n = \langle l, u_n \rangle$.

This paper explores algorithms for finding linear functionals associated with the convection - diffusion problem

$$-\varepsilon \Delta U + b \cdot \nabla U = f(x, y), \text{ in } \Omega = (0,1)^2, \ U = g \text{ on } \partial\Omega. \tag{1.5}$$

The goal is to create a fair comparison of various methods that solve (1.5). Central difference and upwind discretizations are used within the Jacobi method and then augmented with two different algorithms for finding given linear functionals, heat flux and mean temperature. This research paper is based on the ideas presented by Dr. William Layton in [1].

1.1 BASIC APPROACH

The basic approach we explore is to solve iteratively the coupled problems

$$Au = f, \text{ and } A^*\phi = l. \tag{1.6}$$

We shall assume that the system from equation (1.1) is solved by some iterative process that is convergent and in which the residual is calculated at each step. Thus, at the end of each step we have available the information u_n and $r_n = b - Au_n$. The following lemma regarding A^* is necessary.

Lemma 1. *Since A is invertible, A^* is invertible, as well.*

Proof. That A invertible is equivalent to saying there exists a matrix B such that $AB = BA = I$. Then $I = I^* = (AB)^* = B^*A^*$. Also, $I = I^* = (BA)^* = A^*B^*$. Hence we have

$$I = B^*A^* = A^*B^*$$

so that there exists a matrix B^* for which the above equality is true. Hence A^* is invertible. □

Thus, consider the solution of the following linear adjoint problem

$$\langle A^*\phi, v\rangle = \langle l, v\rangle\,,\ \forall v \in \mathbf{R}^n, \tag{1.7}$$

where $\langle\cdot,\cdot\rangle$ is an inner product on $\mathbf{R}^n$ and A^* is the adjoint of A with respect to the chosen inner product. Then for some $l, v \in \mathbf{R}^n$, (1.7) is simply

$$A^*\phi = l \tag{1.8}$$

Lemma 2. *The linear system $A^*\phi = l$ is equivalent to: find $\phi \in \mathbf{R}^n$ satisfying $\langle A^*\phi, v\rangle = l^{tr}v$ for all $v \in \mathbf{R}^n$.*

Proof. For $l, v \in \mathbf{R}^n$, assume $\langle\cdot,\cdot\rangle$ is the Euclidean inner product. Then,

$$\begin{aligned}
\langle A^*\phi, v\rangle &= l^{tr}v,\ \forall\ v \in \mathbf{R}^n \\
\langle A^*\phi, v\rangle &= \langle l, v\rangle\,,\ \forall\ v \in \mathbf{R}^n \\
\langle A^*\phi - l, v\rangle &= 0,\ \forall\ v \in \mathbf{R}^n
\end{aligned}$$

Choosing $v = A^*\phi - l$ yields

$$\langle A^*\phi - l, A^*\phi - l\rangle = 0$$

so that $\|A^*\phi - l\|^2 = 0$, so that $A^*\phi = l$. Equivalence is shown by merely reversing these steps. □

In equation (1.7), choose $v = e_n = u - u_n$, so that

$$\langle A^*\phi, v\rangle = \langle A^*\phi, e_n\rangle = \langle l, e_n\rangle = \langle l, u\rangle - \langle l, u_n\rangle .$$

Solving for $\langle l, u\rangle = l(u)$ yields

$$\begin{aligned} l(u) &= \langle A^*\phi, e_n\rangle + \langle l, u_n\rangle \\ &= \langle \phi, Ae_n\rangle + \langle l, u_n\rangle \\ &= \langle \phi, r_n\rangle + \langle l, u_n\rangle \\ &= l(u_n) + \langle \phi, r_n\rangle \end{aligned} \tag{1.9}$$

Thus, for a linear functional $l(\cdot)$, it is possible to calculate $l(u)$ exactly by

$$l(u) = l(u_n) + \langle \phi, r_n\rangle \tag{1.10}$$

provided that ϕ can be calculated. Clearly, solving equation (1.8) is as difficult as the original one. However, from equation (1.10), it is clear that the complete solution ϕ is not needed—we only need the component of ϕ in the direction of r. Further, since the solution ϕ satisfies an adjoint equation if, for example, the bi-conjugate gradient method were used to solve the original problem, then the adjoint problem could be solved at negligible extra cost or storage. Thus, let $\phi, r_n \in \mathbf{R}^n$. Let $\beta = \left\{r_n, r_1^\perp, \ldots, r_{n-1}^\perp\right\}$ be a basis for $\mathbf{R}^n$, since there exist $n-1$ mutually orthogonal vectors to r_n. Write $\phi = \gamma r_n + \gamma_1 r_1^\perp + \cdots + \gamma_{n-1} r_{n-1}^\perp$. Then the inner product satisfies

$$\langle \phi, r_n\rangle = \left\langle \gamma r_n + \gamma_1 r_1^\perp + \cdots + \gamma_{n-1} r_{n-1}^\perp, r\right\rangle = \gamma \langle r_n, r_n\rangle \tag{1.11}$$

and only the value of the coefficient γ need be approximated. With the substitution, reconsider the dual problem for ϕ.

$$l = A^*\phi = A^*(\gamma r_n + \psi) \tag{1.12}$$

where $\psi \in span\{r_n\}^\perp$. The simplest and most reasonable approximation is to choose γ by a descent step in the direction of the u-residual r_n:

$$\gamma_n = \underset{v=\gamma r_n}{\arg\min} \|l - A^*v\| \tag{1.13}$$

In other words, we can calculate γ_n by minimizing $\|l - A^*v\|^2$ while letting $v = \gamma_n r_n$.

$$\begin{aligned} \|l - A^*v\|^2 &= \langle l - A^*(\gamma_n r_n), l - A^*(\gamma_n r_n)\rangle \\ &= \langle l, l\rangle - 2\gamma_n \langle l, A^* r_n\rangle + \gamma_n^2 \langle A^* r_n, A^* r_n\rangle \\ &= \|l\|^2 - 2\gamma_n \langle l, A^* r_n\rangle + \gamma_n^2 \|A^* r_n\| \\ &= g(\gamma_n) \end{aligned}$$

Then

$$\begin{aligned} g'(\gamma_n) &= 2\|A^* r_n\|^2 \gamma_n - 2\langle A^* r_n, l\rangle = 0 \\ &\Rightarrow \quad \gamma_n = \frac{\langle A^* r_n, l\rangle}{\langle A^* r_n, A^* r_n\rangle} \end{aligned} \tag{1.14}$$

Using this value for γ_n yields the following algorithm:

Algorithm 3. *Post-Processing*

Given u_n

Compute:

$r_n = f - Au_n$

$v = A^* r_n$

$\gamma_n = \frac{\langle A^* r_n, l\rangle}{\langle A^* r_n, A^* r_n\rangle}$

$l_n = \langle l, u_n\rangle + \gamma_n \langle r_n, r_n\rangle$

The second algorithm we consider is simply a global descent method for the adjoint equation where the descent direction is, as above, taken to be the residual in the original equation (1.1).

Algorithm 4. *Descent*

Given $u_n, \phi_n, s_n = l - A^*\phi_n$

Calculate u_{n+1}

Compute:

$r_{n+1} = f - Au_{n+1}$

$v_{n+1} = A^* r_{n+1}$

$\gamma_{n+1} = \frac{\langle A^* r_{n+1}, s_n\rangle}{\langle A^* r_{n+1}, A^* r_{n+1}\rangle}$

$\phi_{n+1} = \phi_n + \gamma_{n+1} r_{n+1}$

$s_{n+1} = l - A^*\phi_{n+1}$

$l_{n+1} = \langle l, u_{n+1}\rangle + \left\langle \phi_{n+1}, r_{n+1}\right\rangle$

This algorithm can be reformulated in various ways. In particular, l_{n+1} can be written as

$$l_{n+1} = l_n + \alpha_n \langle s_n, p_n\rangle + \gamma_n \langle r_{n+1}, r_{n+1}\rangle, \quad \alpha_n \in \mathbf{R}, \quad p_n \in \mathbf{R}^N$$

which can become extremely useful if using the bi-conjugate gradient method as an iterative solver. This is due to the fact that many of the above parameters are calculated at each iterative step.

Proposition 5. *Let u_n satisfy*

$$u_{n+1} = u_n + \alpha_n p_n, \ \alpha_n \in \mathbf{R}, \ p_n \in \mathbf{R}^N \tag{1.15}$$

and let ϕ_n satisfy

$$\phi_{n+1} = \phi_n + \gamma_{n+1} r_{n+1}, \ \gamma_{n+1} \in \mathbf{R}, \ r_{n+1} \in \mathbf{R}^N \tag{1.16}$$

Let $r_n = f - Au_n$, and let $s_n = l - A^*\phi_n$. Then,

$$s_{n+1} = s_n - \gamma_{n+1} A^* r_{n+1} \tag{1.17}$$

and if $\gamma_{n+1} = \frac{\langle A^* r_{n+1}, s_n\rangle}{\langle A^* r_{n+1}, A^* r_{n+1}\rangle}$, and $l_{n+1} = \langle l, u_{n+1}\rangle + \left\langle \phi_{n+1}, r_{n+1}\right\rangle$, then

$$l_0 = \langle l, u_0\rangle + \gamma_0 \langle r_0, r_0\rangle, \tag{1.18}$$

$$l_{n+1} = l_n + \alpha_n \langle s_n, p_n\rangle + \gamma_n \langle r_{n+1}, r_{n+1}\rangle \tag{1.19}$$

Proof. Let equations (1.15) and (1.16) hold. Let $r_{n+1} = r_n - \alpha_n A p_n$, and let $s_n = l - A^* \phi_n$. Let $\gamma_{n+1} = \frac{\langle A^* r_{n+1}, s_n \rangle}{\langle A^* r_{n+1}, A^* r_{n+1} \rangle}$ and $l_{n+1} = \langle l, u_{n+1} \rangle + \langle \phi_{n+1}, r_{n+1} \rangle$. Then

(i)

$$\begin{aligned} s_{n+1} &= l - A^* \phi_{n+1} \\ &= l - A^* (\phi_n + \gamma_{n+1} r_{n+1}) \\ &= l - A^* \phi_n - \gamma_{n+1} A^* r_{n+1} \\ &= s_n - \gamma_{n+1} A^* r_{n+1} \end{aligned}$$

(ii) If $\psi_0 \in span\{r_0\}^{\perp}$, then

$$\begin{aligned} l_0 &= \langle l, u_0 \rangle + \langle \phi_0, r_0 \rangle \\ &= \langle l, u_0 \rangle + \langle \gamma_0 r_0 + \psi_0, r_0 \rangle \\ &= \langle l, u_0 \rangle + \gamma_0 \langle r_0, r_0 \rangle \end{aligned}$$

(iii) Since $l_{n+1} = \langle l, u_{n+1} \rangle + \langle \phi_{n+1}, r_{n+1} \rangle$, then,

$$\begin{aligned} l_{n+1} &= \langle l, u_n + \alpha_n p_n \rangle + \langle \phi_n + \gamma_{n+1} r_{n+1}, r_n - \alpha_n A p_n \rangle \qquad (1.20) \\ &= \langle l, u_n \rangle + \alpha_n \langle l, p_n \rangle + \langle \phi_n, r_n - \alpha_n A p_n \rangle + \gamma_{n+1} \langle r_{n+1}, r_{n+1} \rangle \\ &= \langle l, u_n \rangle + \langle \phi_n, r_n \rangle + \alpha_n \langle l, p_n \rangle - \alpha_n \langle \phi_n, A p_n \rangle + \gamma_{n+1} \langle r_{n+1}, r_{n+1} \rangle \\ &= l_n + \alpha_n \langle l - A^* \phi_n, p_n \rangle + \gamma_{n+1} \langle r_{n+1}, r_{n+1} \rangle \\ &= l_n + \alpha_n \langle s_n, p_n \rangle + \gamma_n \langle r_{n+1}, r_{n+1} \rangle . \qquad (1.21) \end{aligned}$$

□

2.0 DISCRETIZATION

In the test problems that follow, the discretization of equation (1.5) creates a $N^2 \times N^2$ linear system by undergoing the following process.

Let the domain Ω be covered by an $(N+2)\times(N+2)$ uniform mesh. Then the meshwidth $h = \frac{1}{N+1}$. Define $x_i = ih$, $y_j = jh$, $f_i^j = f(x_i, y_j)$, and $g_i^j = g(x_i, y_j)$. Let $U_i^j = U(x_i, y_j)$ be the approximate solution to (1.5). Then, by using central discretization, equation (1.5) becomes

$$
\begin{aligned}
-\varepsilon\Delta U + b\cdot\nabla U &= f(x,y) && (2.1)\\
&= -\varepsilon\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right) + b_1\frac{\partial U}{\partial x} + b_2\frac{\partial U}{\partial y} = f(x,y) && (2.2)\\
&= -\varepsilon\left(\frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2} + \frac{U_i^{j+1} - 2U_i^j + U_i^{j-1}}{h^2}\right) && (2.3)\\
&\quad + b_1\left(\frac{U_{i+1}^j - U_{i-1}^j}{2h}\right) + b_2\left(\frac{U_i^{j+1} - U_i^{j-1}}{2h}\right) && (2.4)\\
&= f_i^j && (2.5)
\end{aligned}
$$

Since the variable U is now a $(N+2)\times(N+2)$ matrix, it is neccessary to convert the matrix into a $N^2 \times 1$ vector by letting $k = N(i-1)+j$ and shifting the known boundary values to the right hand side. In the matrix U, $4N+4$ boundary values are given by the boundary condition $U_i^j = g_i^j$ on $\partial\Omega$. These values, multiplied by the appropriate coefficients, are then added to f to create a new right hand side vector $\tilde{f}$. Let A be a $N^2 \times N^2$ coefficient matrix. The linear system is $AU = \tilde{f}$.. The matrix A is sparse, with at most five entries

per row. In general, the k^{th} line of A will have nonzero components in the following places:

$$\begin{aligned} A_{k,k} &= \frac{4\varepsilon}{h^2} \\ A_{k,k-1} &= \frac{-\varepsilon}{h^2} - \frac{b_2}{2h} \\ A_{k,k-N} &= \frac{-\varepsilon}{h^2} - \frac{b_1}{2h} \\ A_{k,k+1} &= \frac{-\varepsilon}{h^2} + \frac{b_2}{2h} \\ A_{k,k+N} &= \frac{-\varepsilon}{h^2} + \frac{b_1}{2h} \end{aligned} \tag{2.6}$$

The matrix A then takes the general form

$$\begin{pmatrix} \ddots & \ddots & & & \ddots & & & \\ \ddots & \ddots & \ddots & & & \ddots & & \\ & \ddots & \ddots & \ddots & & & \ddots & \\ & & \ddots & \ddots & \ddots & & & \ddots \\ \ddots & & & \ddots & \ddots & \ddots & & \\ & \ddots & & & \ddots & \ddots & \ddots & \\ & & \ddots & & & \ddots & \ddots & \ddots \\ & & & \ddots & & & \ddots & \ddots \end{pmatrix}$$

By observing the nonzero entries of A, it is easy to see that A is symmetric if the vector $\mathbf{b}$ is the zero vector.

An upwind differencing scheme is also used within the scope of the paper, where the second order partial derivatives in the convection term are discretized as above, and the first order partial derivatives in the diffusion term are discretized as follows:

$$b \cdot \nabla T = b_1 \frac{\partial T}{\partial x} + b_2 \frac{\partial T}{\partial y} \tag{2.7}$$

$$= b_1 \begin{cases} \frac{U_{i+1}^j - U_i^j}{h}, & \text{if } b_1 < 0 \\ \frac{U_i^j - U_{i-1}^j}{h}, & \text{if } b_1 \geq 0 \end{cases} \tag{2.8}$$

$$+ b_2 \begin{cases} \frac{U_i^{j+1} - U_i^j}{h}, & \text{if } b_2 < 0 \\ \frac{U_i^j - U_i^{j-1}}{h}, & \text{if } b_2 \geq 0 \end{cases} \tag{2.9}$$

Again, the equation $k = N(i-1) + j$ is used so that $U_k = U_i^j$ in order to create an $N^2 \times N^2$ system.

3.0 A NUMERICAL COMPARISON

We shall consider the problem of estimating two linear functionals in a thermal convection-diffusion problem. Thus, let $U(x, y)$ be the solution of

$$-\varepsilon \Delta U + \mathbf{b} \cdot \nabla U = f(x, y), \text{ in } \Omega = (0, 1)^2, \ U = g \text{ on } \partial\Omega, \tag{3.1}$$

where ε is the diffusion coefficient, b is the convection field, and $f(x, y)$ is the heat source or sink. An interesting functional associated with equation (3.1) is the average temperature:

$$L(U) = \iint_{\Omega} U(x, y) dx dy. \tag{3.2}$$

In 2-D, the the average temperature is calculated discretely by

$$L(U) = \frac{1}{N^2} \sum_{k=1}^{N^2} U_k.$$

Written in terms of dot products, $L(U)$ becomes

$$L(U) = \begin{pmatrix} \frac{1}{N^2} & \cdots & \frac{1}{N^2} \end{pmatrix} \begin{pmatrix} U_1 \\ \vdots \\ U_{N^2} \end{pmatrix} = \begin{pmatrix} \frac{1}{N^2} \\ \vdots \\ \frac{1}{N^2} \end{pmatrix}^{tr} \cdot \begin{pmatrix} U_1 \\ \vdots \\ U_{N^2} \end{pmatrix}. \tag{3.3}$$

The vector l is then $\frac{1}{N^2} \begin{pmatrix} 1 & \cdots & 1 \end{pmatrix}^{tr}$ so that $L(U) = \langle l, U \rangle$.

Another linear functional addressed in this paper is the heat flux. Let Γ denote one face of the flow domain with outward unit normal $\hat{n}$. The problem is then, given diffusion

coefficient ε, the convection field $\mathbf{b}$, heat source or sink $f(x,y)$, and temperature on the boundary $g(x,y)$, find

$$L_F(U) = \int_\Gamma \varepsilon \bigtriangledown U \cdot \hat{n} ds.$$

In the case where $f(x,y) = \mathbf{0}$, $g(x,y) = x$, and the hot wall is the side $x = 1$, heat flux is represented using the equation

$$L_F(U) = \int_0^1 \varepsilon \frac{\partial U}{\partial x}(1,y) dy. \tag{3.4}$$

Lemma 6. *Using equation (3.1) with $f(x,y) = \mathbf{0}$, $g(x,y) = x$, $\varepsilon = 1$, and the hot wall side $x = 1$, the heat flux from equation (3.4) is equivalent to*

$$L_F(U) = h + N + \langle l, U \rangle \tag{3.5}$$

where

$$l = \begin{cases} 0, \, k \leq N^2 - N \\ -1, \, k > N^2 - N \end{cases}. \tag{3.6}$$

Proof. Let $\Psi = \varepsilon \frac{\partial U}{\partial x}(1,y)$ and $\Psi_j = \frac{1 - U_N^j}{h}$. Then, by the Trapezoid Rule, equation (3.4)

becomes

$$\begin{aligned}
L_F(U) &= \int_0^1 \varepsilon \frac{\partial U}{\partial x}(1,y)dy \\
&= \int_0^1 \Psi dy \\
&= \sum_{j=1}^{N+1} \frac{h}{2}(\Psi_{j-1} + \Psi_j) \\
&= \frac{h}{2}\left(\sum_{j=0}^{N} \Psi_j + \sum_{j=1}^{N+1} \Psi_j\right) \\
&= \frac{h}{2}\left(\Psi_0 + 2\sum_{j=1}^{N} \Psi_j + \Psi_{N+1}\right) \\
&= \frac{h}{2}\left[(\Psi_0 + \Psi_{N+1}) + 2\sum_{j=1}^{N} \Psi_j\right] \\
&= \frac{h}{2}(1+1) + h\sum_{j=1}^{N} \frac{1 - U_N^j}{h} \qquad (3.7) \\
&= h + \sum_{j=1}^{N}(1 - U_N^j) \\
&= h + N - \sum_{j=1}^{N} U_N^j.
\end{aligned}$$

For $k = (i-1)N + j$, $l_k = -1$ for $k > N^2 - N$. Then

$$L_F(U) = h + N + \langle l, U \rangle$$

□

Since $L_F(U)$ is affine but not linear, the iterative methods calculate the linear functional $\langle l, U \rangle$ to the desired tolerance and then sum $h + N + \langle l, U \rangle$.

3.1 SYMMETRIC CASE ($\mathbf{B} = \mathbf{0}$)

When the diffusion coefficient vector $\mathbf{b} = \mathbf{0}$, the matrix A becomes symmetric. This results in an ideal case for study because equation (3.1) simplifies to the Model Poisson Problem in 2-D.

3.1.1 Calculating the Average Temperature

In equation (3.1), let $\varepsilon = 1$, $\mathbf{b} = [0,0]$, $f(x,y) = \mathbf{0}$, $g(x,y) = \mathbf{x}$. As stated above, equation (3.2) is the average temperature, which in 2-D can be simplified discretely into $l(U) = \frac{1}{N^2}\sum_{k=1}^{N^2} U_k$. In this symmetric case, equation (3.1) can be solved exactly on the unit square, obtaining $l(U) = \frac{1}{2}$. This is true since the solution to the equation $\triangle u = 0$, $u = x$ on $\partial\Omega$, for $\Omega = [0,1] \times [0,1]$ is $u = x$. Hence, the average temperature can be calculated by solving $l(U) = \int_0^1 \int_0^1 x dx dy = \frac{1}{2}$. The exact solution of $l(U)$ is useful in finding the true error when implementing both the post-processing and descent algorithms.

Choose the Jacobi Method as an iterative solver, and let the convergence tolerance $tol = 0.001$ and the initial guess $Uinitial = \mathbf{0}$. To determine the effectiveness of augmenting the iterative method with algorithms (3) and (4), we need to compare the number of iterations needed for the temperature U to converge with the number of iterations needed for the average temperature L to converge when using the augmented algorithm. In Matlab, the codes that calculate the functional L and the temperature U are denoted *JacobiPP* and *JacobiDes*.

At this point, it is necessary to comment on the stopping criteria used within the trials of this paper. The stopping criteria was initially designed to be used when the exact value of U could not be calculated due to the large size of the associated coefficient matrix A. However, since this project is designed to test the usefulness of the algorithms only, and the sizes of the systems are small enough to allow Matlab to solve the linear system exactly, it became necessary to solve for the exact numerical solution of U. This can be done by solving the system $AU = f$, and Matlab computes this solution easily using the command `Uexact=A\f`. The error for U at each iteration is stored in a vector the size of the maximum number of iterations. In Matlab, the error for U is stored by `error(iter)=norm(U-Uexact)` where U is the value of the solution calculated using the Jacobi algorithm at each iteration. *JacobiPP* and *JacobiDes* find the number of iterations needed for U to converge using the following stopping criteria:

```
if error(iter) < tol & error(iter-1) > tol
```

```
erroriter=error(iter)
end
```

The error for the average value L is calculated by using the same method as above. First, the exact value of L, denoted $Lexact$, is calculated by performing the dot product $\langle l, Uexact \rangle$ where l is the vector representing the linear functional and $Uexact$ is the solution to the linear system $AU = f$. The post-processing or descent algorithm is then applied within the Jacobi iteration to update the value of L. The error for L at each iteration is also stored in a vector the size of the maximum number of iterations. In Matlab, the error for L is stored by `errorL(iter)=abs(L-Lexact)` where L is the value of the linear functional at each step of the Jacobi iteration. The same stopping criteria as above is then applied to $errorL(iter)$:

```
if errorL(iter) < tol & errorL(iter-1) > tol
    errorLiter = errorL(iter)
end
```

For both *error(iter)* and the *errorL(iter)* the error of the previous iteration is tested in the stopping criteria to ensure adequate convergence. Since the augmented *Jacobi* algorithms iterate until the maximum number of iterations, denoted *max_it*, the stopping criteria records both the iterations at which the error in U and the error in L drop below the tolerance level. However, this stopping criteria also allows us to plot the errors versus the iterations well beyond the points of convergence.

Using the conditions and stopping criteria from above, *JacobiPP* was used to solve for the average temperature using many different meshwidths h. As we let h decrease, we increased the maximum number of iterations to accomodate the larger system. From this series of tests, an interesting phenomenon occured which lends itself to more research than is within the scope of this paper. For even values of N between $N = 6$ and $N = 94$, convergence to L using the post-processing algorithm occurs between 4.000 and 5.131 times faster than convergence to U using the regular Jacobi method. However, for odd values of N between $N = 7$ and $N = 95$, L only converges between 1.5 and 2.004 times as fast as U. Using the data from the even case, it appears that as the value of N increases and hence the system

gets larger, *JacobiPP* will continue to converge to the average functional L faster than *Jacobi* will converge to the solution U. Additionally, the data also suggests that as the value of N increases, the ratio between the number of iterations for *Jacobi* to converge and the number of iterations for *JacobiPP* to converge will also increase, implying that for very large values of N, *JacobiPP* will converge considerably faster than *Jacobi*. This observation is similar for odd values of N with the exception that the ratio is much smaller. Tables 1 and 2 provide the data using both even and odd values of N. In the following tables, h is the mesh size of the system, N is the number of unknown values on one side of the matrix U, max_it is the maximum number of iterations, $iter$ is the number of iterations required for U to converge to tolerance, $iterPP$ is the number of iterations required for L to converge to tolerance, and $ratio$ is the value acquired from dividing $iter$ and $iterPP$.

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/8	7	200	76	18	4.000
1/11	10	400	203	50	2.060
1/15	14	800	394	93	4.237
1/23	22	1400	977	221	4.421
1/31	30	2000	1835	401	4.576
1/47	46	6000	4408	922	4.781
1/63	62	9000	8158	1656	4.926
1/95	94	20000	19306	3763	5.131

Table 1: Symmetric case using JacobiPP and even N, average functional

The following semilogarithmic plots show the behavior of the *error* for U and the *errorL* for L. The *error* for U is shown using a solid blue line, and the *errorL* for L is shown using a dashed red line. From figures 1, 2, 3 and 4, the obvious difference in slopes shows the significantly greater convergence rates of the even values of N. Figures 1 and 2 represent small values of N while figures 3 and 4 represent large values of N.

Using the same intitial conditions and stopping criteria, the average temperature is also

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/8	7	200	102	68	1.500
1/12	11	400	245	152	1.612
1/16	15	800	453	269	1.684
1/24	23	1400	1069	603	1.773
1/32	31	2000	1962	1070	1.834
1/48	47	6000	4608	2400	1.920
1/64	63	10000	8433	4259	1.980
1/96	95	25000	19735	9564	2.064

Table 2: Symmetric case using JacobiPP and odd N, average functional

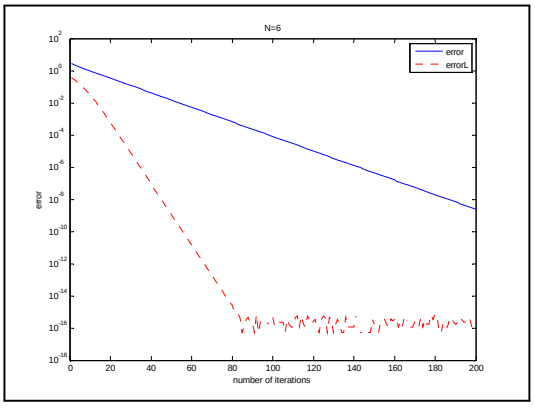

Figure 1: JacobiPP and small, even N, average functional

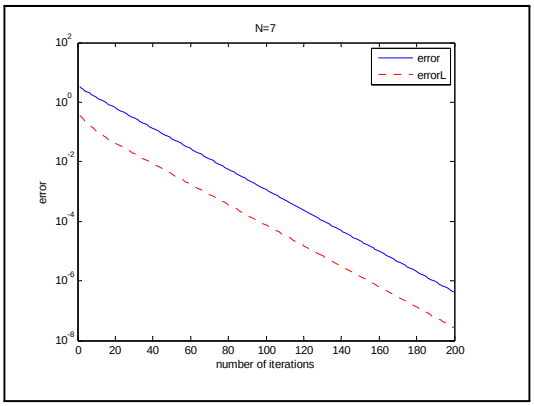

Figure 2: JacobiPP and small, odd N, average functional

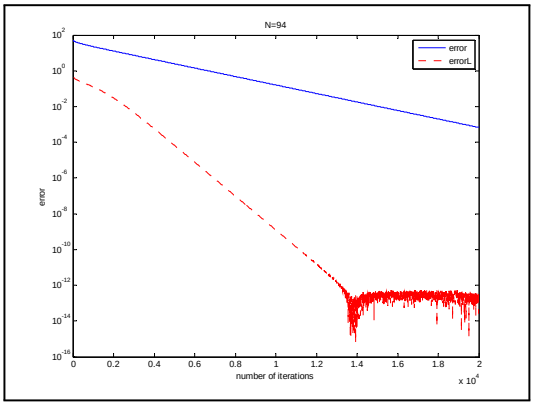

Figure 3: JacobiPP and large, even N, average functional

calculated using the Jacobi method augmented with the descent algorithm, denoted *JacobiDes*. The descent algorithm is ideally used with an iterative solver such as the conjugate gradient method in which the variables of equation (1.19) are already in use; however, it is augmented with the Jacobi method here in order to provide a fair comparison between the two algorithms. Tables 3 and 4 show the data collected for both even and odd values of N. In both cases, the trials show that the average functional L converges between three and four times faster than the solution U. As the meshwidth decreases, the ratio of convergence between L and U increases, which suggests the possibilty of L converging much faster for very small meshwidths.

It is easy to see from figures 5, 6, 7, and 8 that regardless of the value of N, the semilogarithmic plots of the error versus number of iterations behave the same. Figures 5 and 6 represent small even and odd values of N while figures 7 and 8 represent large even and odd values of N.

3.1.2 Calculating the Heat Flux

Again, using equation (3.1) with $\varepsilon = 1$, $\mathbf{b} = [0, 0]$, $f(x, y) = \mathbf{0}$, $g(x, y) = \mathbf{x}$, the heat flux is calculated using equation (3.5) within the Jacobi method augmented with the algorithms. In Matlab, the programs are denoted *JacobiPPflux* and *JacobiDflux*. The program first solves for the exact values of U and L by using the commands `Uexact=A\f` and `Lexact=dot(l,Uexact)+h+N` where l is the same as in equation (3.6), h is the meshwidth, and N is the number of unknown points along one side of the matrix U. Recall that these values of *Uexact* and *Lexact* are used when testing the errors of U and L, respectively.

JacobiPPflux runs through the maximum number of iterations, dentoed *max_it*. The number of iterations needed for U to converge is stored in *iter* and the number of iterations needed for L to converge is stored in $iterPP$. Table 5 and 6 show the number of iterations needed for convergence of U and L as well as the ratio between the iterations. Table 5 uses even values of N whereas Table 6 uses odd values of N. For even values of N, the ratio between *iter* and $iterPP$ rises from 2.303 to 2.921, and for odd values of N, the ratio rise from 1.214 to 1.578 suggesting that once again, the use of even values of N provides faster

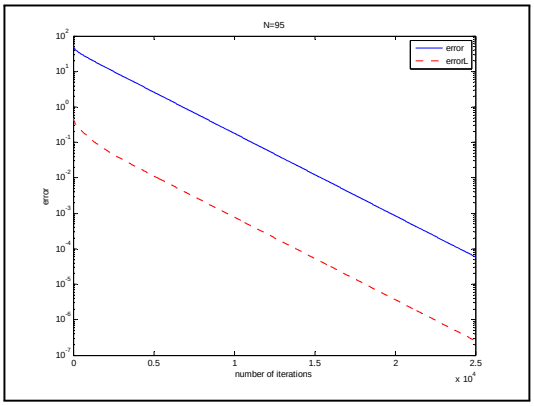

Figure 4: JacobiPP and large, odd N, average functional

			JacobiDescent		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/7	6	200	76	25	3.040
1/11	10	400	203	64	2.172
1/15	14	800	394	120	3.283
1/23	22	1400	977	282	3.465
1/31	30	2000	1835	513	3.577
1/47	46	6000	4408	1177	3.745
1/63	62	9000	8158	2114	3.859
1/95	94	20000	19306	4801	4.021

Table 3: Symmetric case using Jacobi Descent and even N, average functional

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/8	7	200	102	33	3.091
1/12	11	400	245	76	3.224
1/16	15	800	453	137	3.307
1/24	23	1400	1069	308	3.471
1/32	31	2000	1962	547	3.587
1/48	47	6000	4608	1228	3.752
1/64	63	10000	8433	2182	3.865
1/96	95	25000	19735	4903	4.025

Table 4: Symmetric case using Jacobi Descent and odd N, average functional

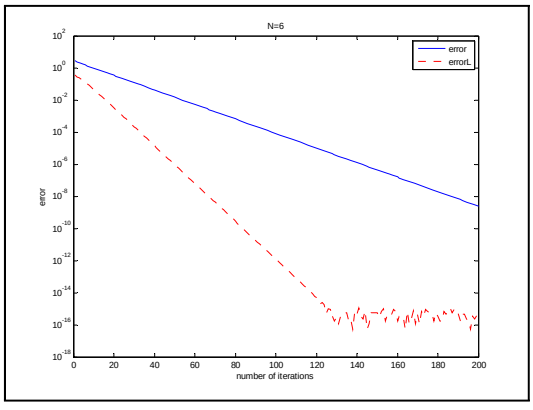

Figure 5: JacobiDes and small, even N, average functional

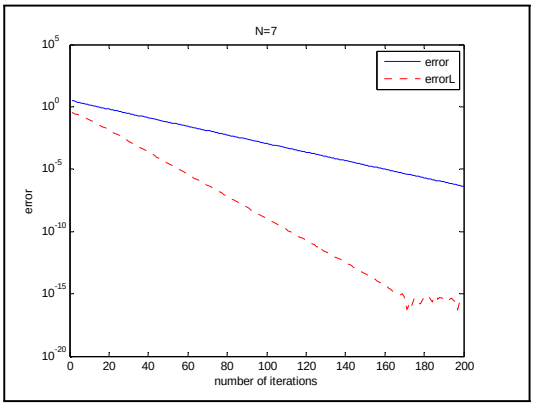

Figure 6: JacobiDes and small, odd N, average functional

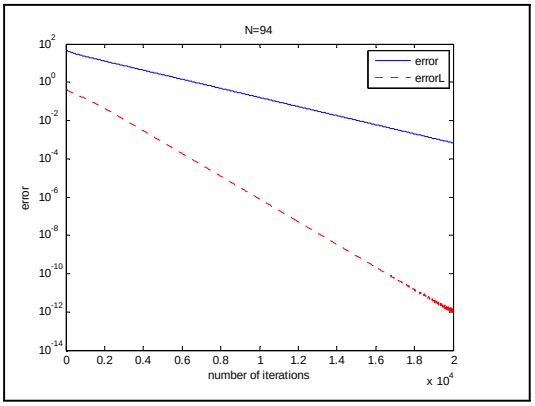

Figure 7: JacobiDes and large, even N, average functional

convergence to L than the use of odd values of N.

Figures 9, 10, 11, and 12 show semilogarithmic plots of error versus the number of iterations where the blue solid line represents the error in U and the red dashed line represents the error in L. Figures 9 and 10 use small values of even and odd N whereas figures 11 and 12 use large values of even and odd N. The figures again clearly show that even values of N produce faster convergence in L than odd values of N, and in all cases, the value of L obtained from the post processing algorithm always converges faster than the value of U. The figures also show dips below the tolerance level prior to actually converging creating the necessity for the particular stopping criteria discussed earlier.

The Jacobi method augmented with the Descent algorithm was also tested in solving for the flux functional. Tables 7 and 8 show the number of iterations needed for U to converge, denoted *iter*, and the number of iterations needed for L to converge, denoted *iterDes*, as well as the ratio of these iterations. The tables show that regardless of whether N is even or odd, the ratio increases from 3.167 to 3.808 meaning that L converges to $Lexact$ more than 3 times faster than U converges to $Uexact$. Additionally, the ratio increases as the meshwidth h decreases, which suggests that *JacobiDflux* will find accurate values of L much faster than U for extremely large systems.

Figures 13, 14, 15, and 16 show semilogarithmic plots of error versus the number of iterations. Again, the solid blue line represents the error in U while the dashed red line represents the error in L. The obvious difference in slopes between the red and blue plots further show the benefits of using the *JacobiDflux*.

3.2 NONSYMMETRIC CASE

While the symmetric case provides positive results, testing the algorithms' effects on the nonsymmetric case are more useful for practical applications. Again consider the convection-diffusion equation (3.1). Using the discretization methods from equations (2.1) and (2.7), a nonzero vector b creates a matrix A which is not symmetric. Solving the system $AU = f$ becomes much more difficult for nonsymmetric systems since many iterative solvers require that the system be symmetric and positive definite. However, since our tests use the iterative

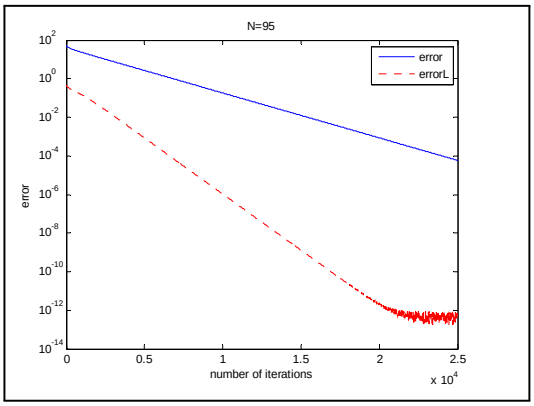

Figure 8: JacobiDes and large odd N, average functional

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/7	6	200	76	33	2.303
1/11	10	400	203	85	2.388
1/15	14	800	394	161	2.447
1/23	22	1400	977	384	2.544
1/31	30	2000	1835	701	2.618
1/47	46	6000	4408	1615	2.730
1/63	62	9000	8158	2905	2.808
1/95	94	20000	19306	6609	2.921

Table 5: Symmetric case using JacobiPP and even N, flux functional

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/8	7	200	102	84	1.214
1/12	11	400	245	192	1.276
1/16	15	800	453	344	1.317
1/24	23	1400	1069	780	1.371
1/32	31	2000	1962	1388	1.414
1/48	47	6000	4608	3124	1.475
1/64	63	10000	8433	5556	1.518
1/96	95	25000	19735	12506	1.578

Table 6: Symmetric case using JacobiPP and odd N, flux functional

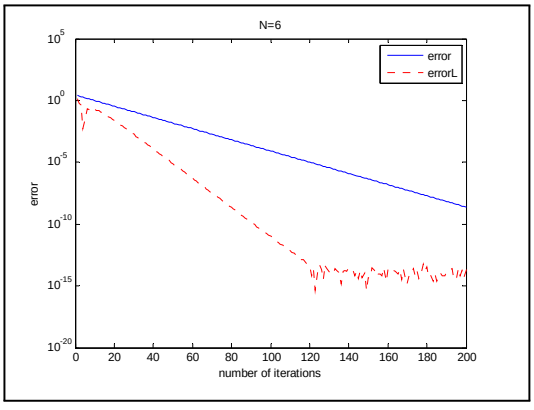

Figure 9: JacobiPP and small even N, flux functional

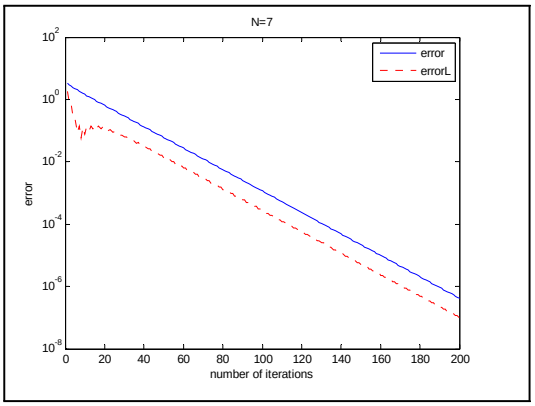

Figure 10: JacobiPP and small odd N, flux functional

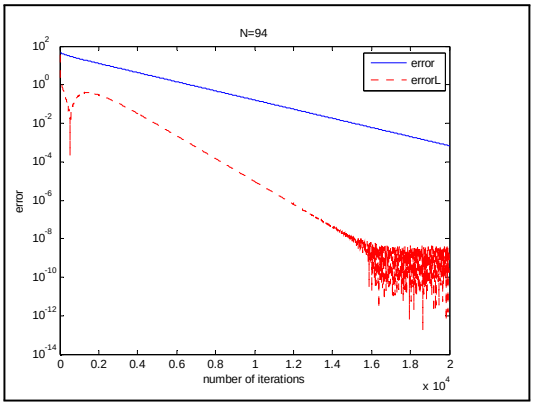

Figure 11: JacobiPP and large even N, flux functional

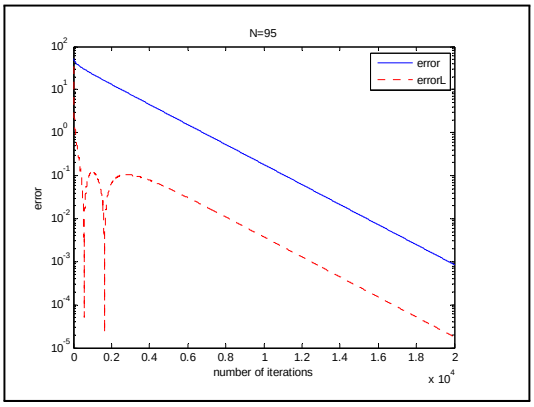

Figure 12: JacobiPP and large odd N, flux functional

			JacobiDescent		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/7	6	200	76	24	3.167
1/11	10	400	203	65	3.123
1/15	14	800	394	123	3.203
1/23	22	1400	977	294	3.323
1/31	30	2000	1835	537	3.417
1/47	46	6000	4408	1239	3.558
1/63	62	9000	8158	2230	3.659
1/95	94	20000	19306	5075	3.804

Table 7: Symmetric case using JacobiDes and even N, flux functional

			JacobiDescent		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/8	7	200	102	32	3.188
1/12	11	400	245	78	3.141
1/16	15	800	453	141	3.213
1/24	23	1400	1069	321	3.330
1/32	31	2000	1962	573	3.424
1/48	47	6000	4608	1293	3.564
1/64	63	9000	8433	2302	3.663
1/96	95	25000	19735	5183	3.808

Table 8: Symmetric case using JacobiDes and odd N, flux functional

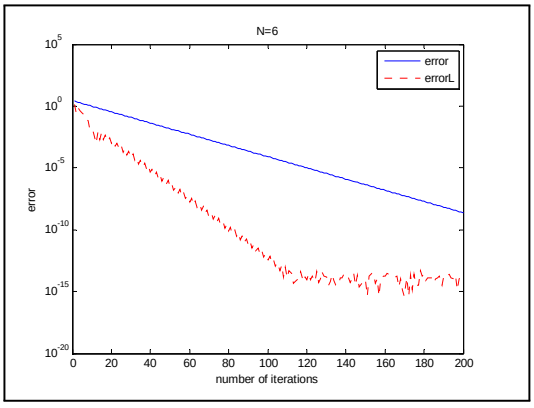

Figure 13: Jacobi Des and small even N, flxu functional

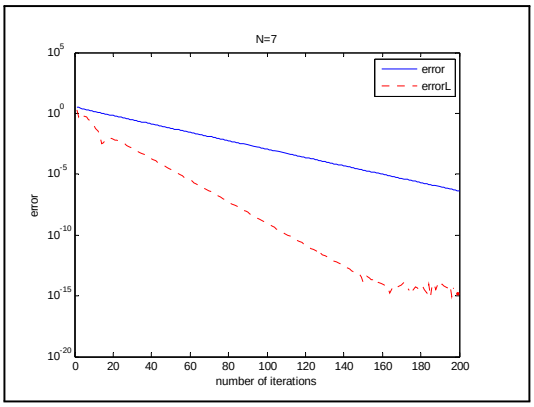

Figure 14: Jacobi Des and small odd N, flxu functional

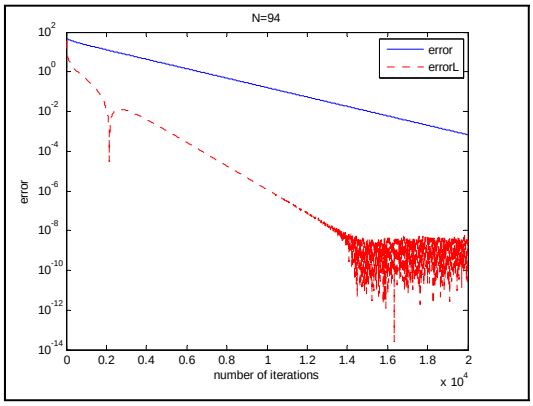

Figure 15: Jacobi Des and large even N, flxu functional

solver *Jacobi,* and *Jacobi* does not require the linear system to be symmetric positive definite, we can solve the nonsymmetric system with few alterations from the symmetric case. In the following nonsymmetric cases, *Jacobi* continues to be augmented with the post processing and descent algorithms, the same stopping criteria are used, and the goal is to compute the average and flux functionals described in the previous chapter. However, for nonsymmetric A, it becomes possible to test the effectiveness of using alternate discretization methods. The following sections test both the central and upwind discretization methods as described in equations (2.1) and (2.7).

3.2.1 Calculating the Average Temperature

In equation (3.1), let $f = 0$, $g = x$, $\varepsilon = 1$, $b = (1, 0)$, and $Uinitial = (0\ 0\ \cdots 0)$. As in the symmetric case, the average temperature is again calculated using equation (3.3).

3.2.1.1 Central Discretization Using the central discretization method from equation (2.1), *JacobiPP* is used to determine the usefulness of augmenting *Jacobi* with the post processing algorithm in the nonsymmetric case. The test was performed just as in the symmetric case using the same values of h and the same tolerance levels. Tables 9 and 10 display the results using even and odd values of N, respectively. Table 9 shows that the ratio between the number of iterations for U to converge and the number of iterations for L to converge range between 2.313 and 3.107. Further, this ratio increases as the meshwidth h decreases, which suggests that the post processing algorithm is effective for extremely small meshwidths. Table 10 shows that for odd values of N, the ratio ranges from 1.547 to 2.122. While these ratios continue to increase as the meshwidth h decreases, the use of even values of N continue to provide better results. However, the results for the nonsymmetric case are far less desirable than the symmetric case.

Figures 17, 18, 19, and 20 show the behavior of the error of U and L versus the number of iterations for small and large values of even and odd N. Note that a visual observation of slopes show they are the same, implying that although *JacobiPP* might provide faster results than *Jacobi* by itself, the speed is only of the order of a constant, which is a less than

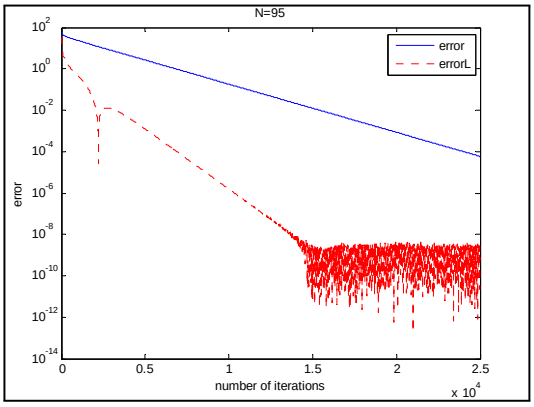

Figure 16: Jacobi Des and large odd N, flxu functional

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/7	6	200	74	32	2.313
1/11	10	400	197	82	2.402
1/15	14	800	383	152	2.520
1/23	22	1400	949	357	2.658
1/31	30	2000	1784	648	2.753
1/47	46	6000	4289	1486	2.886
1/63	62	9000	7941	2666	2.979
1/95	94	20000	18803	6052	3.107

Table 9: NonSymmetric case using JacobiPP, central, and even N; average

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/8	7	200	99	64	1.547
1/12	11	400	238	144	1.653
1/16	15	800	440	255	1.726
1/24	23	1400	1039	572	1.816
1/32	31	2000	1908	1014	1.882
1/48	47	6000	4483	2273	1.972
1/64	63	9000	8208	4034	2.035
1/96	95	20000	19220	9058	2.122

Table 10: NonSymmetric case using JacobiPP, central, and odd N; average

optimal result.

The program *JacobiDes* is also applied to the nonsymmetric case $b = (1, 0)$ in order to solve for the average functional. Tables 11 and 12 provide the results for this test. In this case, whether N is even or odd makes no difference in rate of convergence of L versus U, as shown by the ratios that range from 1.762 to 2.297. The results show that as the meshwidth h decreases, the ratio between the number of iterations for U to converge and the number of iterations for L to converge increases.

Figures 21 through 24 are similar to the previous trial, implying that L converges faster for smaller meshwidths by order of a constant.

3.2.1.2 Upwind Discretization In the following trial, the upwind discretization from (2.7), (2.8), and (2.9) are used to create the matrix A and the right hand side f. *JacobiPP* is then applied to calculate the average functional using $b = [1, 0]$. Tables 13 and 14 show that even values of N yield better convergence rates with the ratio ranging from 2.500 to 3.126. The use of odd values of N provides increasing ratios; however L is converging only between 1.539 and 2.121 times faster than U.

Figures 25 through 28 again show a trend similar to the previous two trials.

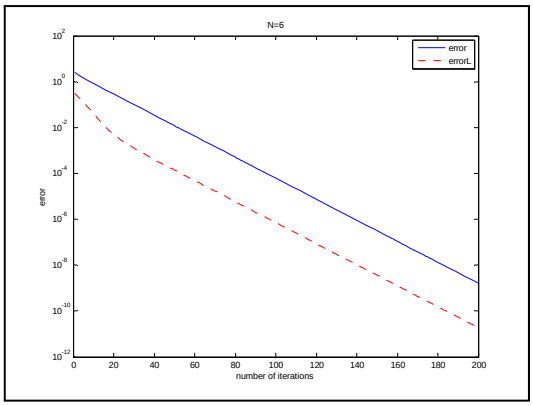

Figure 17: Jacobi PP and small even N, average functional

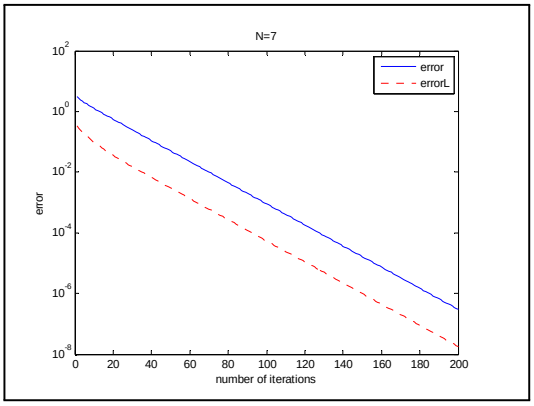

Figure 18: Jacobi PP and small odd N, average functional

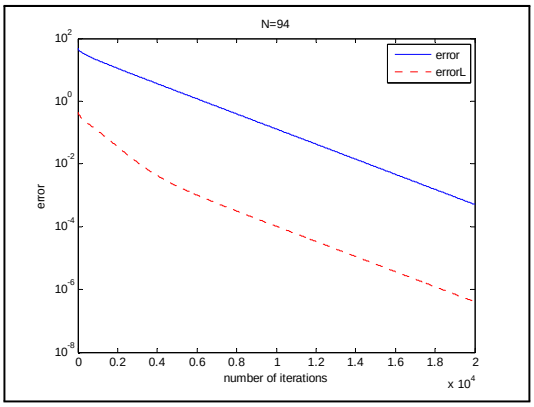

Figure 19: JacobiPP and large even N, average functional

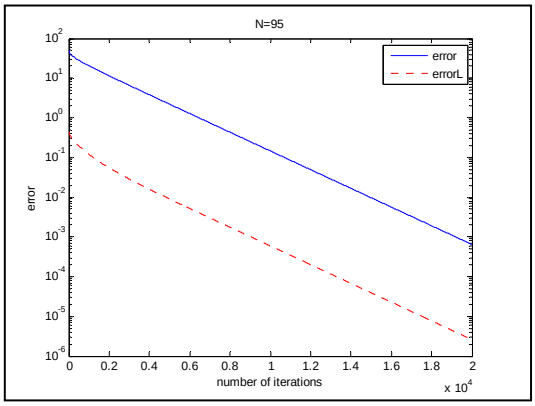

Figure 20: JacobiPP and large odd N, average functional

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/7	6	200	74	42	1.762
1/11	10	400	197	108	1.824
1/15	14	800	383	202	1.896
1/23	22	1200	949	478	1.985
1/31	30	2000	1784	870	2.051
1/47	46	5000	4289	2003	2.141
1/63	62	8000	7941	3601	2.205
1/95	94	20000	18803	8193	2.295

Table 11: NonSymmetric case using JacobiDes, central, and even N; average

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/8	7	200	99	56	1.768
1/12	11	400	238	128	1.859
1/16	15	800	440	230	1.913
1/24	23	1200	1039	520	1.998
1/32	31	2000	1908	927	2.058
1/48	47	5000	4483	2089	2.146
1/64	63	9000	8208	3716	2.209
1/96	95	20000	19220	8367	2.297

Table 12: NonSymmetric case using JacobiDes, central, and odd N, average

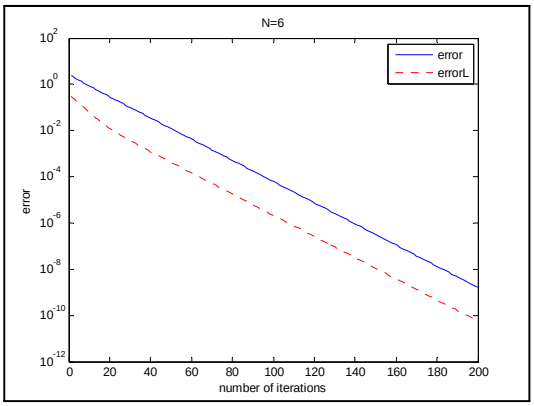

Figure 21: Jacobi Des and small even N, average functional

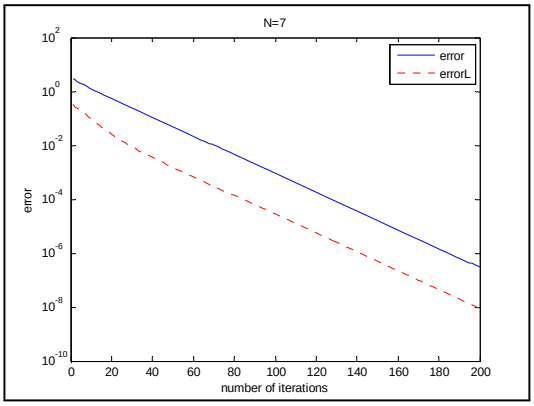

Figure 22: JacobiDes and small odd N, average functional

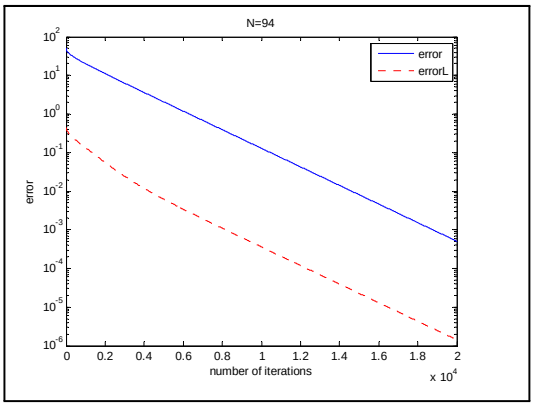

Figure 23: JacobiDes and large even N, average functional

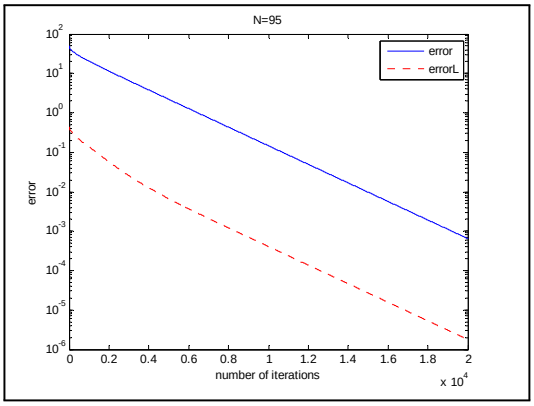

Figure 24: JacobiDes and large odd N, average functional

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/7	6	200	75	30	2.500
1/11	10	400	198	78	2.539
1/15	14	800	384	147	2.612
1/23	22	1200	952	349	2.728
1/31	30	2000	785	291	2.698
1/47	46	5000	4294	1470	2.921
1/63	62	8000	7948	2644	3.006
1/95	94	20000	18813	6019	3.126

Table 13: NonSymmetric case using JacobiPP, upwind, and even N; average

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/8	7	200	100	65	1.539
1/12	11	400	239	145	1.648
1/16	15	800	441	257	1.716
1/24	23	1200	1041	574	1.814
1/32	31	2000	1911	1016	1.881
1/48	47	5000	4488	2277	1.971
1/64	63	9000	8215	4039	2.034
1/96	95	20000	19230	9066	2.121

Table 14: NonSymmetric case using JacobiPP, upwind, and odd N; average

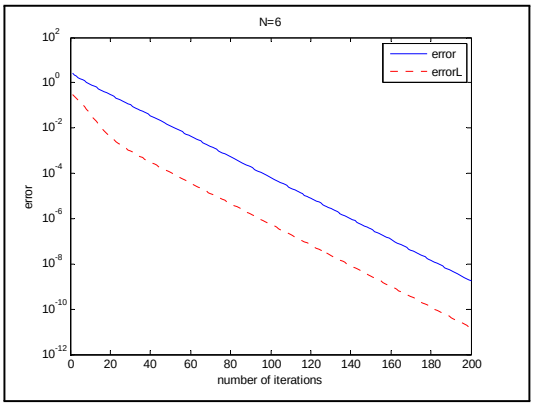

Figure 25: JacobiPP and small even N, average functional

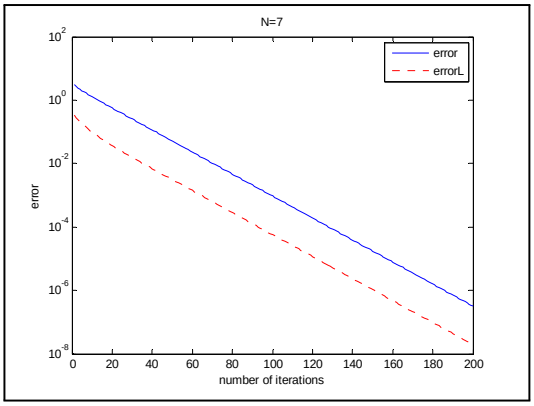

Figure 26: JacobiPP and small odd N, average functional

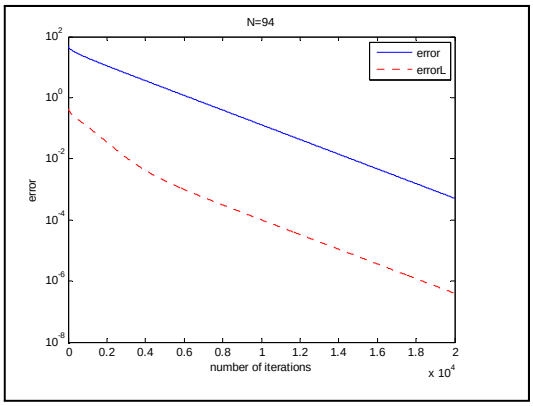

Figure 27: JacobiPP and large even N, average functional

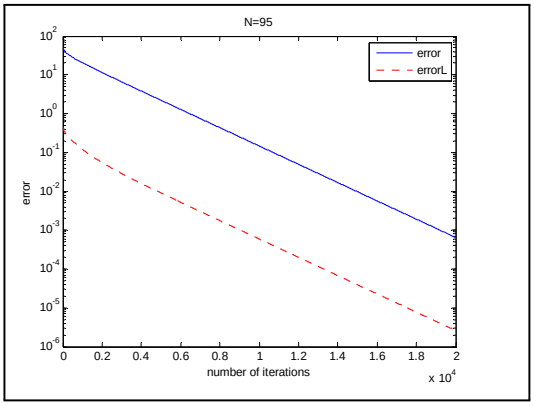

Figure 28: JacobiPP and large odd N, average functional

JacobiDes is also applied to the average functional using upwind discretization, and tables 15 and 16 provide the results of this trial. In this case, the use of even or odd N does not have an effect on the ratio of the iterations, and L converges between 1.829 and 2.303 times faster than U.

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/7	6	200	75	41	1.829
1/11	10	400	198	106	1.868
1/15	14	800	384	199	1.930
1/23	22	1200	952	474	2.008
1/31	30	2000	1787	865	2.066
1/47	46	5000	4294	1995	2.152
1/63	62	8000	7948	3591	2.213
1/95	94	20000	18813	8178	2.300

Table 15: NonSymmetric case using JacobiDes, upwind, and even N; average

Figures 29 through 32 again show similar results.

3.2.2 Calculating the Heat Flux

In equation (3.1), let $f = 0$, $g = x$, $\varepsilon = 1$, $b = (1, 0)$, and $Uinitial = (0\ 0\ \cdots 0)$. As in the symmetric case, the heat flux is calculated using equation (3.5).

3.2.2.1 Central Discretization Using central discretization, Jacobi and the post processing algorithm are paired to solve for the heat flux when $b = (1, 0)$. The meshwidths h and tolerance levels are the same as the previous section. Tables 17 and 18 show that for even values of N, the ratio between $iter$ and $iterPP$ decreases from 1.175 to 1.082 as h decreases, and for odd values of N, the ratio decreases from 1.547 to 1.157.

The graphs representing this data are too similar to previous figures to include here.

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/8	7	200	100	55	1.818
1/12	11	400	239	126	1.897
1/16	15	800	441	227	1.943
1/24	23	1200	1041	516	2.017
1/32	31	2000	1911	922	2.073
1/48	47	5000	4488	2081	2.157
1/64	63	9000	8125	3706	2.217
1/96	95	20000	19230	8351	2.303

Table 16: NonSymmetric case using JacobiDes, upwind, and odd N; average

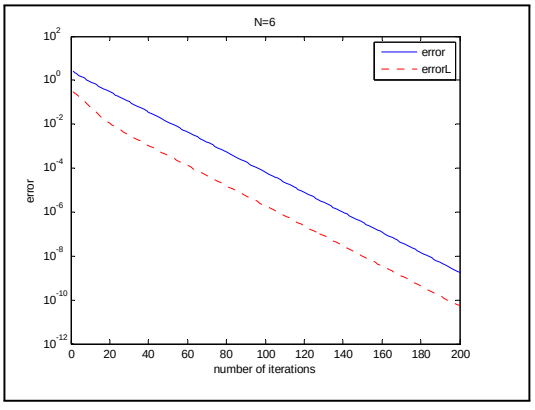

Figure 29: JacobiDes and small even N, average functional

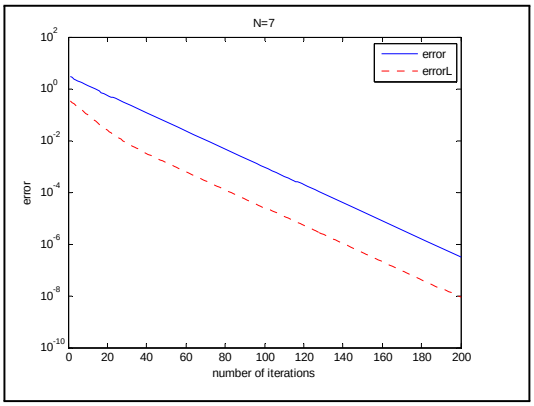

Figure 30: JacobiDes and small odd N, average functional

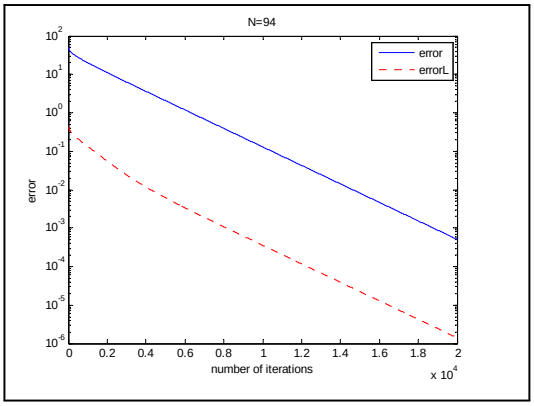

Figure 31: JacobiDes and large even N, average functional

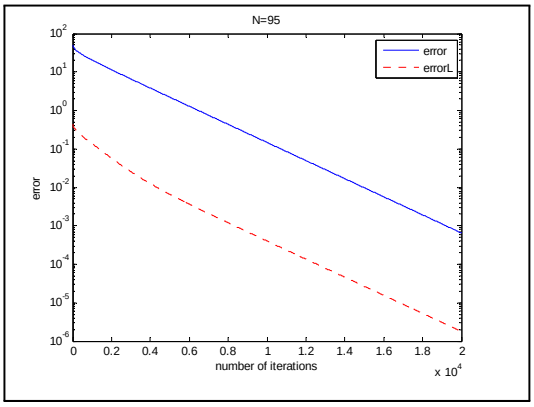

Figure 32: JacobiDes and large odd N, average functional

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/7	6	200	74	63	1.175
1/11	10	400	197	174	1.132
1/15	14	800	383	342	1.112
1/23	22	1400	949	859	1.105
1/31	30	2000	1784	1624	1.099
1/47	46	5000	4289	3931	1.091
1/63	62	8000	7941	7306	1.087
1/95	94	20000	18803	17376	1.082

Table 17: NonSymmetric case using JacobiPP, central, and even N; flux

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/8	7	200	99	64	1.547
1/12	11	400	238	155	1.536
1/16	15	800	440	325	1.354
1/24	23	1400	1039	829	1.253
1/32	31	2000	1908	1566	1.218
1/48	47	5000	4483	3776	1.187
1/64	63	10000	8208	7001	1.172
1/96	95	20000	19220	16611	1.157

Table 18: NonSymmetric case using JacobiPP, central, and odd N; flux

Tables 19 and 20 provide results when the descent algorithm is used instead of post processing. In this case, regardless of whether N is even or odd, the ratio between $iter$ and $iterPP$ decreases from 1.850 to 1.326 as h decreases.

Figures 33 though 36 depict the data from Tables 19 and 20. Note that aside from the growing dip in the $errorL$ line, there is littler difference in the figures, suggesting that the descent algorithm accelerates convergence poorly regardless of whether N is even or odd and regardless of the value of h.

3.2.2.2 Upwind Discretization The heat flux is also calculated using upwind discretization from equations (2.7), (2.8), and (2.9). Tables 21 and 22 were created by augmenting the post processing algorithm with the Jacobi solver. As in the previous cases regarding flux, the ratio between $iter$ and $iterPP$ continue to decrease as h decreases suggesting that the $JacobiPP$ program is not an effective tool to solve for the heat flux.

Figures 37, 38, 39, and 40 show the results from the above tables. From the figures, note that the odd case converges slightly faster than the even case.

Finally, the descent method is used to find the heat flux using upwind discretization.

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/7	6	200	74	40	1.850
1/11	10	400	197	123	1.602
1/15	14	800	383	251	1.526
1/23	22	1400	949	645	1.451
1/31	30	2000	1784	1262	1.414
1/47	46	5000	4289	3122	1.373
1/63	62	8000	7941	5874	1.352
1/95	94	20000	18803	14171	1.327

Table 19: NonSymmetric case using JacobiDes, central, even N; flux

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/8	7	200	99	57	1.737
1/12	11	400	238	150	1.587
1/16	15	800	440	290	1.517
1/24	23	1400	1039	719	1.445
1/32	31	2000	1908	1353	1.410
1/48	47	10000	4483	3267	1.372
1/64	63	10000	8208	6076	1.351
1/96	95	20000	19220	14492	1.326

Table 20: NonSymmetric case using JacobiDes, central, and odd N; flux

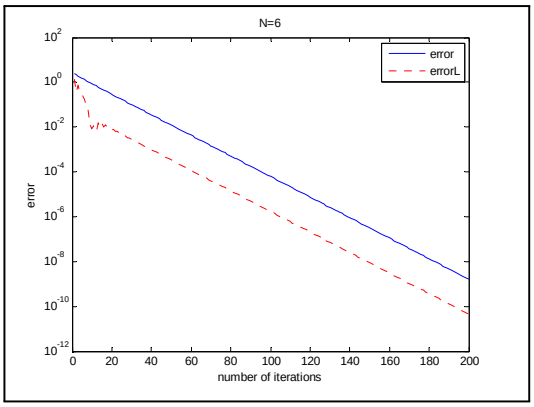

Figure 33: JacobiDes and small even N, flux functional

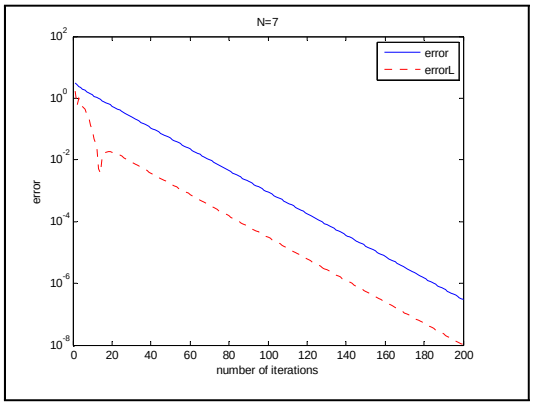

Figure 34: JacobiDes and small odd N, flux functional

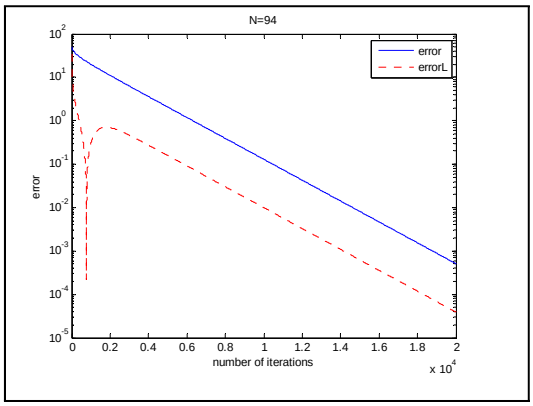

Figure 35: JacobiDes and large even N, flux functional

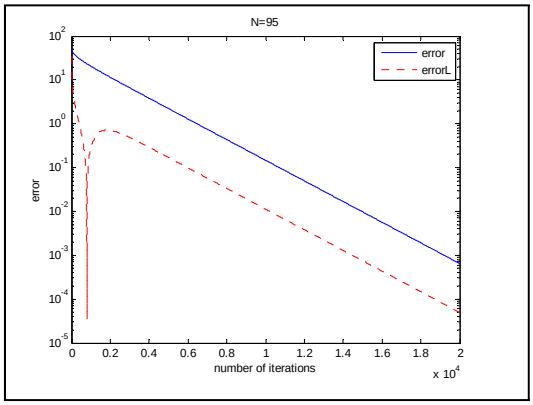

Figure 36: JacobiDes and large odd N, flux functional

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/7	6	200	75	62	1.210
1/11	10	400	198	172	1.151
1/15	14	800	384	340	1.129
1/23	22	1400	952	855	1.114
1/31	30	2000	1787	1619	1/104
1/47	46	5000	4294	3923	1.095
1/63	62	8000	7948	7295	1.090
1/95	94	20000	18813	17361	1.084

Table 21: NonSymmetric case using JacobiPP, upwind, and even N; flux

			JacobiPP		ratio
h	N	max_it	iter	iterPP	iter/iterPP
1/8	7	200	100	68	1.471
1/12	11	400	239	141	1.695
1/16	15	800	441	317	1.391
1/24	23	1400	1041	819	1.271
1/32	31	2000	1911	1555	1.229
1/48	47	5000	4488	3763	1.192
1/64	63	10000	8215	6984	1.176
1/96	95	20000	19230	16587	1.159

Table 22: NonSymmetric case using JacobiPP, upwind, and odd N; flux

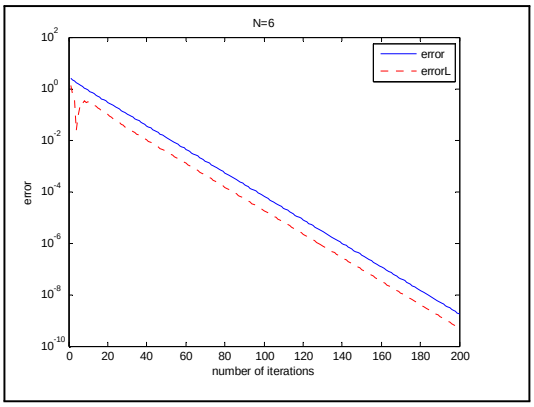

Figure 37: JacobiPP and small even N, flux functional

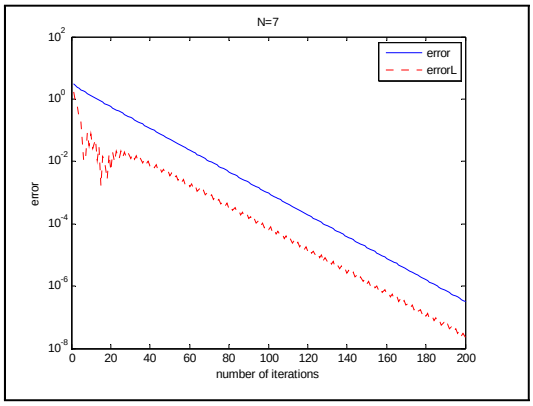

Figure 38: JacobiPP and small odd N, flux functional

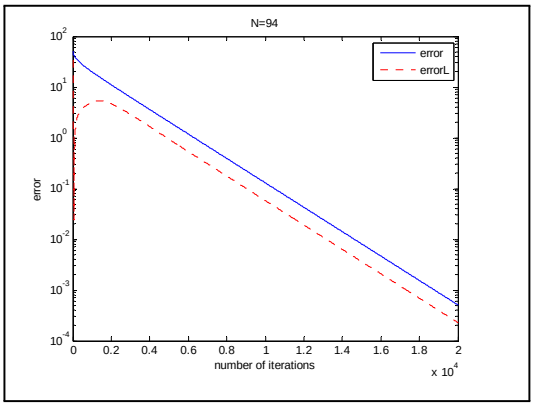

Figure 39: JacobiPP and large even N, flux functional

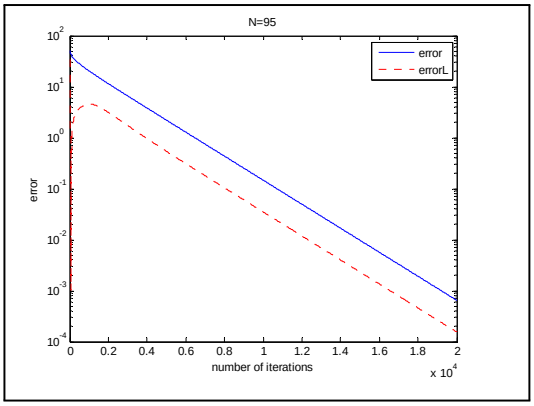

Figure 40: JacobiPP and large odd N, flux functional

Tables 23 and 24 show that regardless of whether N is even or odd, the ratio between $iter$ and $iterDes$ decreases from 1.875 to 1.328.

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/7	6	200	75	40	1.875
1/11	10	400	198	122	1.623
1/15	14	800	384	250	1.536
1/23	22	1400	952	652	1.460
1/31	30	2000	1787	1259	1.419
1.47	46	5000	4294	3117	1.378
1/63	62	8000	7948	5867	1.355
1/95	94	20000	18813	14161	1.329

Table 23: NonSymmetric case using JacobiDes, upwind, and even N; flux

Figures 41 through 44 visually represent the data from Tables 23 and 24.

			JacobiDes		ratio
h	N	max_it	iter	iterDes	iter/iterDes
1/8	7	200	100	57	1.754
1/12	11	400	239	150	1.593
1/16	15	800	441	289	1.526
1/24	23	1400	1041	717	1.452
1/32	31	2000	1911	1350	1.416
1/48	47	5000	4488	3262	1.376
1/64	63	10000	8215	6069	1.354
1/96	95	20000	19230	14482	1.328

Table 24: NonSymmetric case using JacobiDes, upwind, and odd N; flux

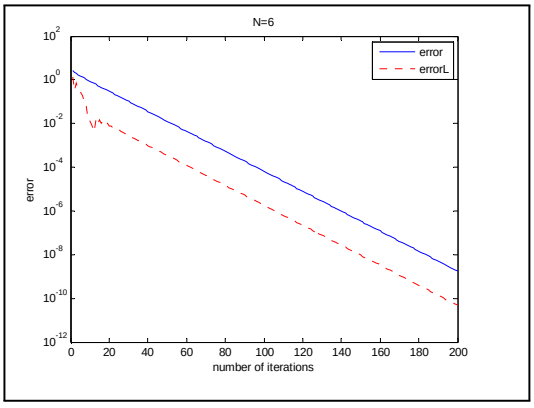

Figure 41: JacobiDes and small even N, flux functional

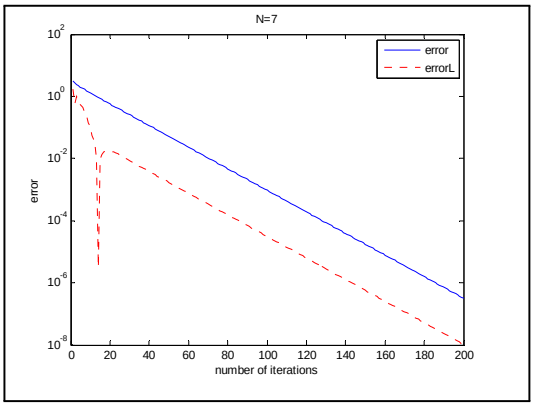

Figure 42: JacobiDes and small odd N, flux functional

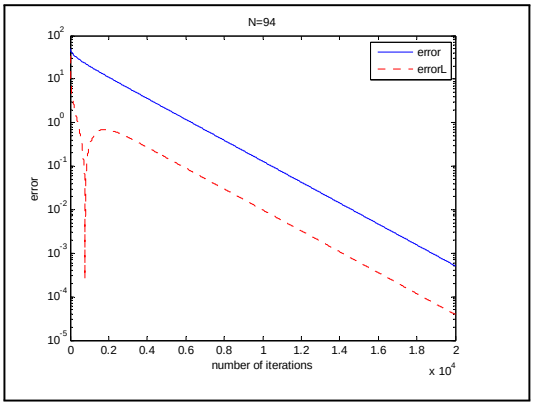

Figure 43: JacobiDes and large even N, flux functional

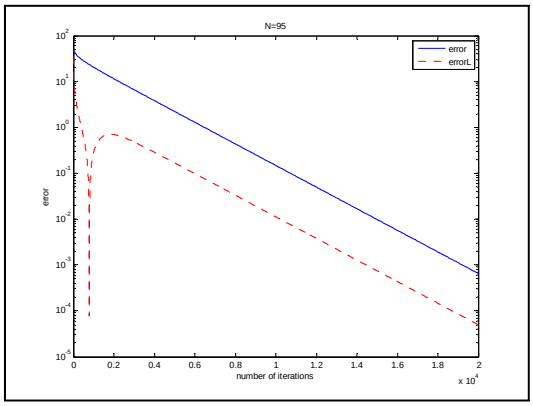

Figure 44: JacobiDes and large odd N, flux functional

4.0 CONCLUSIONS

The robustness of the post processing and descent algorithms were explored by augmenting these algorithms with the Jacobi method to solve for the average and flux functionals. The data showed that the usefulness of the algorithms depended largely on which functional was being calculated and in some cases, whether the size N of the matrix U was even or odd. For example, when the post processing algorithm was used with Jacobi to calculate the average functional for the symmetric case and N was even, the functional L converged between 4-5 times faster than the convergence of U. In contrast, when the post processing algorithm was used with Jacobi to calculate the flux functional for the symmetric case and N was odd, the functional L converged only 1.2-1.5 times faster than the convergence of U. Further, when any algorithm was used to solve for the flux functional in the nonsymmetric case, the rate of convergence to L decreased as the meshwidth h decreased.

By far, the best results came from augmenting the post processing algorithm to the Jacobi method in solving for the average functional, and most specifically when N was even. In this case, the average functional was calculated about four times faster than solving for the solution U to the desired tolerance. However, other interesting results occurred as well.

First, data from the symmetric case showed increasing ratios as the meshwidths decreased, which implies that the algorithms will continue to accelerate the Jacobi method in finding the average and flux functionals for large systems. This is also true for finding the average functional in the nonsymmetric case. However, when calculating the flux in the nonsymmetric case, the ratios consistently decreased regardless of the discretization or algorithm used.

The results, however inconsistent, raised many questions.

- It was surprising to see that even and odd N demonstrated significantly different convergence behaviors in some, but not all, cases. Further study of this phenomenon might reveal the potential for even greater convergence rate gains.
- In the nonsymmetric case, calculating the average functional produced increasing ratios and calculating the flux functional produced decreasing ratios. Further research could show why the choice of the functional appears to have an effect on the usefulness of the algorithms.
- In these trials, only the Jacobi method was used as an iterative solver. Other iterative methods could easily accelerate the algorithms, particularly the bi-conjugate gradient method.
- The choice of stopping criteria played a large role in the development of the programs used. The use of alternate stopping criteria could easily provide faster results.
- The use of preconditioning never played a role in this research. The adequate use of preconditioners could have a large effect on the convergence of L.

It is obvious that this research has sparked many questions regarding the acceleration of convergence of linear functionals. In this sense, further research is necessary to provide conclusive evidence that the post processing and descent algorithms do indeed accelerate this convergence. However, this paper has opened the doors to an exciting area of research with the possibility of finding many useful tools in the area of computational mathematics.

BIBLIOGRAPHY

[1] Layton, William, "Calculating Functionals of Solutions of Large, Sparse Systems".

[2] Barrett, R. et al. "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", SIAM, 1994.

[3] Kelley, C.T.,. "Frontiers in Applied Mathematics", SIAM, 1995.

[4] Van der Vorst,Henk A., "Lecture Notes on Iterative Methods", 1994.

[5] Freund, Roland W., Gene H. Golub, and Noel M. Nachtigal, "Iterative Solution of Linear Systems", 1991.

[6] Atkinson, Kendall E., "An Introduction to Numerical Analysis", Wiley, 1989.

[7] Strang, Gilbert, "Introduction to Applied Mathematics", Wellesley-Cambridge Press, 1986.

[8] Brenner, Susanne C., and L. Ridgway Scott, "The Mathematical Theory of Finite Element Methods", Springer-Verlag, 1994.

[9] Saad, Yousif and Kesheng Wu, "Design of an Iterative Solution Module for a Parallel Sparse Matrix Library", 1995.

[10] Weisstein et al., "Conjugate Gradient Method", http://mathworld.wolfram.com /ConjugateGradientMethod.html

[11] Dongarra, Jack, "BiConjugate Gradient (BiCG)", http://www.netlib.org /linalg/html_templates/node32.html

[12] Weisstein et al., "Conjugate Gradient Method on the Normal Equations", http://mathworld.wolfram.com /ConjugateGradientMethodontheNormalEquations.html

[13] http://www.netlib.org/linalg/old_html_templates.html

[14] Sussman, Mike, "Math 2071: Lab 5: Solving Linear Systems", http://www.math.pitt.edu /~sussmanm/2071Spring06/lab05/index.html

APPENDIX

MATLAB CODE

```
function A=SparseAmatrix1(N,epsilon,b)
    % The function A=SparseAmatrix1[N] creates a sparse (N^2)x(N^2) matrix A
    % from the Convection-Diffusion problem using the central differencing
    % scheme.

    % Trisha Butler
    % July 6, 2006

    h=1/(N+1);
    A=sparse(N^2,N^2);

    for m=1:N^2
      A(m,m)=4*epsilon/(h^2);
      if m-1>0&(mod(m-1,N)~=0)
        A(m-1,m)=(-epsilon/(h^2))+(b(2)/(2*h));
      end
      if (m+1<=N^2)&(mod(m+1,N)~=1)
        A(m+1,m)=(-epsilon/(h^2))-(b(2)/(2*h));
      end
      if m-N>0
        A(m-N,m)=(-epsilon/(h^2))+(b(1)/(2*h));
```

```
  end
  if m+(N)<=N^2
    A(m+N,m)=(-epsilon/(h^2))-(b(1)/(2*h));
  end
end
```

```
function f=RHS1(N,epsilon,b)
% The function RHS1(N,epsilon,b) creates the right hand side
% in the linear system A*U=f

% Trisha Butler
% July 7, 2006

h = 1/(N+1);

xx=linspace(0,1,N+2);
x=xx(2:N+1);
yy=linspace(0,1,N+2);
y=yy(2:N+1);

% Set the boundary conditions:
TBottom = x;                % x = 0
TLeft = zeros(size(y));     % y = 0
TTop = x;                   % x = 1
TRight = ones(size(y));     % y = 1

% Initialize the RHS vector:
f=zeros(N^2,1);

%left boundary
j=1;
for i=1:N
   k=(j-1)*N+i;
   f(k)=f(k)+ ( (b(1)/(2*h)) + (epsilon/(h^2)) )*TLeft(i);
end

% bottom boundary
i=1;
for j=1:N
   k=(j-1)*N+i;
```

```
   f(k)=f(k)+( (b(2)/(2*h)) + epsilon/(h^2)) *TBottom(j);
end

% top boundary
i=N;
for j=1:N
   k=(j-1)*N+i;
   f(k)= f(k)+( epsilon/(h^2) - (b(2)/(2*h)) ) *TTop(j);
end

% right boundary
j=N;
for i=1:N
   k=(j-1)*N+i;
   f(k)=f(k)+ ( epsilon/(h^2) - (b(1)/(2*h)) ) *TRight(i);
end
```

```
function [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiPP(U,N,max_it,tol,epsilon,b )
% The function [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
% JacobiPP(U,N,max_it,tol,epsilon,b)
% iteratively solves the linear system AU=f for U using the Jacobi Method and then estimates the average
% linear functional using the post processing algorithm.

% Trisha Butler
% April 25, 2006

% input  U            REAL initial guess vector
%        N            INTEGER dimension
%        max_it       INTEGER maximum number of iterations
%        tol          REAL error tolerence
%        epsilon      REAL diffusion coefficient epsilon
%        b            REAL convection field b

% output  L           REAL linear functional approximation
%         U           REAL solution vector
%         error       REAL error norm
%         errorL      REAL error for linear functional
%         iter        INTEGER number of iterations performed
%         erroriter   INTEGER # of iterations for Jacobi convergence
%         errorLiter  INTEGER # of iterations for JacobiPP convergence
%         flag        INTEGER: 0 = solution found to tolerance
%                              1 = no convergence given max_it
%         L0          REAL vector of dot products dot(l,U)
%         term        REAL vector of correction terms alpha*dot(r,r)

A=SparseAmatrix1(N,epsilon,b);
f=RHS1(N,epsilon,b);
```

```
Uexact = A\f;
iter = 0;
flag = 0; % initialization

bnrm2 = norm(f);
if ( bnrm2 == 0.0 )
   bnrm2 = 1.0;
end

r = f - A*U;
error = norm( r )/bnrm2;
if (error < tol)
   return
end

D = diag(diag(A)); % matrix splitting
rd = (1.0)./diag(A);
B =-( A - D );

l = (1/N^2)*ones(N^2,1); % initialize l
Lexact = dot(l,Uexact);
error = zeros(max_it,1);
errorL = zeros(max_it,1);
erroriter = 0;
errorLiter = 0;

for iter = 1:max_it % begin iteration
 U_1 = U;
 r_1 = r;
 U = rd .* (B*U + f); % update approximation
 r = f - A*U;
```

```
error(iter) = norm(U - Uexact);
if (error(iter) < tol) & (error(iter-1) > tol)
  erroriter = iter;
end




p = A'*r;
alpha = (dot(p,l))/(dot(p,p));
term(iter) = alpha * dot(r,r);
L0(iter) = dot(l,U);
L = L0(iter) + term(iter);

errorL(iter) = abs(L-Lexact);

if (errorL(iter) < tol) & (errorL(iter-1) > tol)
  errorLiter = iter;
end

end
if (error > tol)
  flag = 1; % no convergence
end
```

```
function [L,max_it,erroriter,errorLiter,ratio] = JacobiPPData(N,epsilon,b,max_it)
% the function [LPP,max_it,erroriter,errorLiter,ratio]=JacobiPPData(N,epsilon,b) fills in
% the table for Jacobi and JacobiPP data and provides a semilog plot of iterations.

% Trisha Butler
% July 5, 2006

% input    N                  INTEGER dimension
%           epsilon           REAL diffusion coefficient
%           b                  REAL convection field
%          max_it              INTEGER maximum number of iterations

% output LPP                  REAL linear functional approximation
%           max_it            INTEGER maximum number of iterations
%           erroriter         INTEGER number of iterations for Jacobi convergence
%           errorLiter        INTEGER number of iterations for JacobiPP convergence
%           ratio              REAL ratio of erroriter to errorLiter

Uinitial = zeros(N^2,1);
tol = .001;

[L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiPP(Uinitial,N,max_it,tol,epsilon,b );
ratio = erroriter/errorLiter;

semilogy(1:iter,error,1:iter,errorL,':red');
legend('error','errorL')
t = sprintf('N=%d',N)
title(t);
xlabel('number of iterations');
ylabel('error');
```

```
function [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiDescent(U,N,max_it,tol,epsilon,b )
% The function [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
% JacobiDescent(U,N,max_it,tol,epsilon,b)
% iteratively solves the linear system AU=f for U using the Jacobi Method and then estimates
the
% average linear functional using the descent algorithm.

% Trisha Butler
% July 7, 2006

% input      U              REAL initial guess vector
%              N              INTEGER dimension
%              max_it         INTEGER maximum number of iterations
%              tol            REAL error tolerence
%              epsilon        REAL diffusion coefficient
%              b              REAL convection field b

% output   L                  REAL linear functional approximation
%              U              REAL solution vector
%              error          REAL error norm
%              errorL         REAL error for linear functional
%              iter           INTEGER number of iterations performed
%              erroriter      INTEGER number of iterations for Jacobi convergence
%              errorLiter     INTEGER number of iterations for
%                                    JacobiDescent convergence
%              flag           INTEGER: 0 = solution found to tolerance
%                                          1 = no convergence given max_it
%              L0             REAL vector of dot products dot(l,U)
%              term           REAL vector of correction terms dot(phi,r)

A=SparseAmatrix1(N,epsilon,b); % REAL matrix
```

```
f=RHS1(N,epsilon,b); % REAL right hand side vector
Uexact = A\f;

phi = zeros(N^2,1);
l = (1/N^2)*ones(N^2,1);    % initialize l
Lexact = dot(l,Uexact);
iter = 0;
flag = 0;      % initialization

bnrm2 = norm(f);
if ( bnrm2 == 0.0 )
   bnrm2 = 1.0;
end

r = f - A*U;
error = norm( r )/bnrm2;
if (error < tol)
    return
end

s = l - A'*phi;
D = diag(diag(A));              % matrix splitting
rd = (1.0)./diag(A);
B =-( A - D );

for iter = 1:max_it                % begin iteration


 U_1 = U;
 r_1 = r;


 U = rd .* (B*U + f); % update approximation
 r = f - A*U;
```

```
error(iter) = norm(U - Uexact);
if (error(iter) < tol) & (error(iter-1) > tol)
    erroriter = iter;
end



p = A'*r;
gamma = (dot(p,s))/(dot(p,p));
phi = phi + gamma*r;
s = l - A'*phi;
term(iter) = dot(phi,r);
L0(iter) = dot(l,U);
L = L0(iter) + term(iter);
errorL(iter) = abs(L - Lexact);
if (errorL(iter) < tol) & (errorL(iter-1) > tol)
    errorLiter = iter;
end
end
if (error > tol)
    flag = 1; % no convergence
end
```

```
function [L,max_it,erroriter,errorLiter,ratio] =
...JacobiDescentData(N,epsilon,b,max_it)
% The function
% [L,max_it,erroriter,errorLiter,ratio]=JacobiDescentData (N,epsilon,b)
% fills in the table for Jacobi and JacobiDescent data and provides a
% semilog plot of iterations.

% Trisha Butler
% July 6, 2006

% input       N                    INTEGER dimension
%               epsilon            REAL diffusion coefficient
%               b                  REAL convection field
%               max_it             INTEGER maximum number of iterations

% output      L                    REAL linear functional approximation
%               max_it             INTEGER maximum number of iterations
%              erroriter           INTEGER number of iterations for Jacobi convergence
%              errorLiter          INTEGER number of iterations for JacobiDescent convergence
%               ratio              REAL ratio of erroriter to errorLiter

Uinitial = zeros(N^2,1);
tol = .001;
[L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiDescent(Uinitial,N,max_it,tol,epsilon,b );
ratio = erroriter/errorLiter;

semilogy(1:iter,error,1:iter,errorL,':red');
legend('error','errorL')
t = sprintf('N=%d',N)
title(t);
xlabel('number of iterations');
ylabel('error');
```

```
function [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiPPflux(U,N,max_it,tol,epsilon,b )
% The function [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
% JacobiPPflux(U,N,max_it,tol,epsilon,b)
% iteratively solves the linear system AU=f for U using the Jacobi Method and then estimates the flux linear
% functional using the post processing algorithm.


% Trisha Butler
% July 5, 2006

% input       U                   REAL initial guess vector
%               N                   INTEGER dimension
%               max_it              INTEGER maximum number of iterations
%               tol                 REAL error tolerence
%               epsilon             REAL diffusion coefficient
%               b                   REAL convection field

% output    L                     REAL linear functional approximation
%             U                   REAL solution vector
%               error               REAL error norm
%               errorL              REAL error for linear functional
%               iter                INTEGER number of iterations performed
%               erroriter           INTEGER number of iterations for Jacobi convergence
%             errorLiter          INTEGER number of iterations for JacobiPP convergence
%               flag                INTEGER: 0 = solution found to tolerance
%                                            1 = no convergence given max_it
%               L0                  REAL vector of dot products dot(l,U)
%               term                REAL vector of correction terms alpha*dot(r,r)

A=SparseAmatrix1(N,epsilon,b);    % REAL matrix
```

```
f=RHS1(N,epsilon,b);                    % REAL right hand side vector
h = 1/(N+1);
Uexact = A\f;                           % Exact solution of the system.
iter = 0;
flag = 0;                               % initialization

bnrm2 = norm(f);
if ( bnrm2 == 0.0 )
     bnrm2 = 1.0;
end

r = f - A*U;
error = norm( r )/bnrm2;
if (error < tol)
    return
end

D = diag(diag(A));                      % matrix splitting
rd = (1.0)./diag(A);
B =-( A - D );
l = flux(N);                            % initialize l
Lexact = dot(l,Uexact) + h + N;

for iter = 1:max_it                     % begin iteration
 U_1 = U;
 U = rd .* (B*U + f);                   % update approximation
 r = f - A*U;


 error(iter) = norm(U - Uexact);        % compute error
 if (error(iter) < tol) & (error(iter-1) > tol)
    erroriter = iter;
 end
```

```
 p = A'*r;
 alpha = (dot(p,l)) / (dot(p,p));
 term(iter) = alpha * (dot(r,r));
 L0(iter) = dot(l,U);
 L = L0(iter) + term(iter) + N + h;


 errorL(iter) = abs(L - Lexact);
 if (errorL(iter) < tol) & (errorL(iter-1) > tol)
     errorLiter = iter;
 end
end

if (error > tol)
    flag = 1;                                  % no convergence
end
```

```
function [L,max_it,erroriter,errorLiter,ratio] = JacobiPPfluxData(N,epsilon,b,max_it)
% the function
% [L,max_it,erroriter,errorLiter,ratio]=JacobiPPfluxData(N,epsilon,b) fills in
% the table for Jacobi and JacobiPPflux and provides a semilog plot of iterations.

% Trisha Butler
% July 6, 2006

% input     N                 INTEGER dimension
%             epsilon         REAL diffusion coefficient
%             b                 REAL convection field
%             max_it            INTEGER maximum number of iterations

% ouput   L                   REAL linear functional approximation
%             max_it            INTEGER maximum number of iterations
%             erroriter         INTEGER number of iterations for Jacobi convergence
%            errorLiter         INTEGER number of iterations for JacobiPPflux convergence
%             ratio             REAL ratio of erroriter to errorLiter

h = 1/(N+1);
Uinitial = zeros(N^2,1);
tol = .001;

[L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiPPflux(Uinitial,N,max_it,tol,epsilon,b );
ratio = erroriter/errorLiter;

semilogy(1:iter,error,1:iter,errorL,':red');
legend('error','errorL')
t = sprintf('N=%d',N)
title(t);
xlabel('number of iterations');
ylabel('error');
```

```
function [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiDflux(U,N,max_it,tol,epsilon,b )
% The function
% [L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
% JacobiDflux(U,N,max_it,tol,epsilon,b)
% iteratively solves the linear system AU=f for U using the Jacobi Method and then estimates
the
% flux linear functional using the descent algorithm.

% Trisha Butler
% July 7, 2006

% input      U              REAL initial guess vector
%            N              INTEGER dimension
%            max_it         INTEGER maximum number of iterations
%            tol            REAL error tolerence
%            epsilon        REAL diffusion coefficient
%            b              REAL convection field

% output   L                REAL linear functional approximation
%            U              REAL solution vector
%            error          REAL error norm
%            errorL         REAL error for linear functional
%            iter           INTEGER number of iterations performed
%            erroriter      INTEGER number of iterations for Jacobi convergence
%            errorLiter     INTEGER number of iterations for JacobiDflux convergence
%            flag           INTEGER: 0 = solution found to tolerance
%                                    1 = no convergence given max_it
%            L0             REAL vector of dot products dot(l,U)
%            term           REAL vector of correction terms dot(phi,r)

A = SparseAmatrix1(N,epsilon,b);
```

```
f = RHS1(N,epsilon,b);
h=1/(N+1);
Uexact = A\f;
phi = zeros(N^2,1);
l = flux(N);
Lexact = dot(l,Uexact) + N + h;
iter = 0;
flag = 0;                              % initialization
bnrm2 = norm(f);
if ( bnrm2 == 0.0 )
      bnrm2 = 1.0;
end

r = f - A*U;
error = norm( r )/bnrm2;
if (error < tol)
    return
end

s = l - A'*phi;
D = diag(diag(A));                              % matrix splitting
rd = (1.0)./diag(A);
B =-( A - D );

for iter = 1:max_it                              % begin iteration
 U_1 = U;
 r_1 = r;
 U = rd .* (B*U + f);                            % update approximation
 r = f - A*U;


 error(iter) = norm(U - Uexact);
```

```
if (error(iter) < tol) & (error(iter-1) > tol)
erroriter = iter;
end



p = A'*r;
gamma = (dot(p,s))/(dot(p,p));
phi = phi + gamma*r;
s = l - A'*phi;



term(iter) = dot(phi,r);
L0(iter) = dot(l,U);
L = L0(iter) + term(iter) + h + N;



errorL(iter) = abs(L - Lexact);
if (errorL(iter) < tol) & (errorL(iter-1) > tol)
    errorLiter = iter;
end
end


if (error > tol)
    flag = 1;                                   % no convergence
end
```

```
function [L,max_it,erroriter,errorLiter,ratio] = JacobiDfluxData(N,epsilon,b,max_it)
% the function
% [L,max_it,erroriter,errorLiter,ratio]=JacobiDfluxData(N,epsilon,b) fills in
% the table for Jacobi and JacobiDflux and provides a semilog plot of iterations.

% Trisha Butler
% July 5, 2006

% input      N                    INTEGER dimension
%              epsilon            REAL diffusion coefficient
%              b                    REAL convection field
%              max_it               INTEGER maximum number of iterations

% output   L                      REAL linear functional approximation
%              max_it               INTEGER maximum number of iterations
%              erroriter            INTEGER number of iterations for Jacobi convergence
%             errorLiter      INTEGER number of iterations for JacobiDescent convergence
%              ratio                REAL ratio of erroriter to errorLiter

h=1/(N+1);
Uinitial = zeros(N^2,1);
tol = .001;
[L,U,error,errorL,iter,erroriter,errorLiter,flag,L0,term]=
...JacobiDflux(Uinitial,N,max_it,tol,epsilon,b );
ratio = erroriter/errorLiter;

semilogy(1:iter,error,1:iter,errorL,':red');
legend('error','errorL')
t = sprintf('N=%d',N)
title(t);
xlabel('number of iterations');
ylabel('error');
```

```
function l=flux(N);

l=zeros(N^2,1);

for i = 1:N^2

   if i > (N^2-N)

        l(i) = -1;

    end

end
```

```
function Uexact = Uexact1(N)
xx=linspace(0,1,N+2);
x=xx(2:N+1);
Uexact = zeros(N^2,1);
for j=1:N
   Uexact(j)=x(j);
end
for k=1:N-1
   for j=k*N+1:(k+1)*N
      U(j)=x(k);
   end
end
```